**x86 Programming I**

CSE 351 Winter 2017

http://xkcd.com/409/
Administrivia

- Lab 2 released!
  - Da bomb!
  - Go to section!

- No Luis OH
  - Later this week
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

OS:
Windows 8
Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
x86 Topics for Today

- Registers
- Move instructions and operands
- Arithmetic operations
- Memory addressing modes
- swap example
What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with % (e.g., %rsi)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
### x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)
### Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Role</th>
<th>Name Origin</th>
<th>16-bit virtual registers (backwards compatibility)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulator</td>
<td>16-bit virtual registers</td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%edx</td>
<td>data base</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%ebx</td>
<td>base pointer</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%esi</td>
<td>source index</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
<td></td>
<td>16-bit virtual registers</td>
</tr>
</tbody>
</table>

- %ax
- %ah
- %al
- %cx
- %ch
- %cl
- %dx
- %dh
- %dl
- %bx
- %bh
- %bl
- %si
- %di
- %sp
- %bp
x86-64 Assembly Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. %xmm1, %ymm2)
  - Come from extensions to x86 (SSE, AVX, ...)
  - Probably won’t have time to get into these 😞
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, ...
  - “Intel”: used by Intel documentation, Intel tools, ...
  - Must know which you’re reading
Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - **Load** data from memory into register
     - \( \%\text{reg} = \text{Mem}[\text{address}] \)
   - **Store** register data into memory
     - \( \text{Mem}[\text{address}] = \%\text{reg} \)

2) Perform arithmetic operation on register or memory data
   - \( c = a + b; \quad z = x << y; \quad i = h \& g; \)

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches
Operand types

- **Immediate**: Constant integer data
  - Examples: $0x400, -533
  - Like C literal, but prefixed with `$`
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other “address modes”

---

<table>
<thead>
<tr>
<th>%rax</th>
<th>%rcx</th>
<th>%rdx</th>
<th>%rbx</th>
<th>%rsi</th>
<th>%rdi</th>
<th>%rsp</th>
<th>%rbp</th>
<th>%rN</th>
</tr>
</thead>
</table>
Moving Data

- **General form:** `mov source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte “byte”

- `movw src, dst`
  - Move 2-byte “word”

- `movl src, dst`
  - Move 4-byte “long word”

- `movq src, dst`
  - Move 8-byte “quad word”
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>movq %rax, (%rdx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

- **Cannot do memory-memory transfer with a single instruction**

  - How would you do it?
Memory vs. Registers

- **Addresses** vs. **Names**
  - 0x7FFFD024C3DC vs. %rdi

- **Big** vs. **Small**
  - ~ 8 GiB vs. (16 x 8 B) = 128 B

- **Slow** vs. **Fast**
  - ~50-100 ns vs. sub-nanosecond timescale

- **Dynamic** vs. **Static**
  - Can “grow” as needed while program runs vs. fixed number in hardware
Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - Maximum of one memory operand
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  - How do you implement "\( r3 = r1 + r2 \)"?

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>( dst = dst + src )</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>( dst = dst - src )</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>( dst = dst \ast src )</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>( dst = dst &gt;&gt; src )</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>( dst = dst &gt;&gt; src )</td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>( dst = dst &lt;&lt; src )</td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>( dst = dst ^ src )</td>
</tr>
<tr>
<td>andq src, dst</td>
<td>( dst = dst &amp; src )</td>
</tr>
<tr>
<td>orq src, dst</td>
<td>( dst = dst</td>
</tr>
</tbody>
</table>

\( a = b + c \)

signed mult
Arithmetic
Logical
(same as salq)
Some Arithmetic Operations

- Unary (one-operand) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq ( \text{dst} )</td>
<td>( \text{dst} = \text{dst} + 1 )  increment</td>
</tr>
<tr>
<td>decq ( \text{dst} )</td>
<td>( \text{dst} = \text{dst} - 1 )  decrement</td>
</tr>
<tr>
<td>negq ( \text{dst} )</td>
<td>( \text{dst} = -\text{dst} )  negate</td>
</tr>
<tr>
<td>notq ( \text{dst} )</td>
<td>( \text{dst} = \sim\text{dst} )  bitwise complement</td>
</tr>
</tbody>
</table>

- See CSPP Section 3.5.5 for more instructions: \( \text{mulq, cqto, idivq, divq} \)
Arithmetic Example

```c
long simple_arith(long x, long y) {
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```
y += x;
y *= 3;
long r = y;
return r;
```
Example of Basic Addressing Modes

```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Understanding swap()

```c
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Registers**
- `%rdi` ↔ `xp`
- `%rsi` ↔ `yp`
- `%rax` ↔ `t0`
- `%rdx` ↔ `t1`

**Memory**

**Swap:***
- `movq (%rdi), %rax`
- `movq (%rsi), %rdx`
- `movq %rdx, (%rdi)`
- `movq %rax, (%rsi)`
- `ret`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### Code

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### swap:

```
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
# Understanding `swap()`

## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>456</td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

## Example Code

```assembly
swap:
    movq (%rdi), %rax        # t0 = *xp
    movq (%rsi), %rdx        # t1 = *yp
    movq %rdx, (%rdi)        # *xp = t1
    movq %rax, (%rsi)        # *yp = t0
    ret
```

---

This slide illustrates the `swap()` function, where values at memory addresses are exchanged using registers and an example code snippet is provided to demonstrate the process.
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

### Code

```
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

### Swap Function

```assembly
swap:
    movq  (%rdi), %rax  # t0 = *xp
    movq  (%rsi), %rdx  # t1 = *yp
    movq  %rdx, (%rdi)  # *xp = t1
    movq  %rax, (%rsi)  # *yp = t0
    ret
```
Memory Addressing Modes: Basic

- **Indirect:** $(R) \quad \text{Mem}[\text{Reg}[R]]$
  - Data in register $R$ specifies the memory address
  - Like pointer dereference in C
  - **Example:** `movq (%rcx), %rax`

- **Displacement:** $D(R) \quad \text{Mem}[\text{Reg}[R] + D]$
  - Data in register $R$ specifies the *start* of some memory region
  - Constant displacement $D$ specifies the offset from that address
  - **Example:** `movq 8(%rbp), %rdx`
Complete Memory Addressing Modes

❖ **General:**

- \( D(R_b, R_i, S) \) \( \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] * S + D] \)
  - \( R_b \): Base register (any register)
  - \( R_i \): Index register (any register except \%rsp)
  - \( S \): Scale factor (1, 2, 4, 8) – *why these numbers?*
  - \( D \): Constant displacement value (a.k.a. immediate)

❖ **Special cases** *(see CSPP Figure 3.3 on p.181)*

- \( D(R_b, R_i) \) \( \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] + D] \) \( (S=1) \)
- \( (R_b, R_i, S) \) \( \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] * S] \) \( (D=0) \)
- \( (R_b, R_i) \) \( \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]] \) \( (S=1, D=0) \)
- \( (, R_i, S) \) \( \text{Mem}[\text{Reg}[R_i] * S] \) \( (R_b=0, D=0) \)
### Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%%rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
D(\text{Rb}, \text{Ri}, S) \rightarrow \text{Mem[Reg[Rb]} + \text{Reg[Ri]} \times S + D]
\]
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(,%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,,%rcx,4)</td>
<td>0xf000 + 0x100*4</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>0xf000*2 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

D(Rb,Ri,S) → Mem[Reg[Rb]+Reg[Ri]*S+D]
Peer Instruction Question

Which of the following statements is TRUE?

(A) The program counter (%rip) is a register that we manually manipulate
(B) There is only one way to compile a C program into assembly
(C) Mem to Mem (src to dst) is the only disallowed operand combination
(D) We can compute an address without using any registers
Summary

- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly instructions have rigid form
  - Operands include immediates, registers, and data at specified memory locations
  - Many instruction variants based on size of data
- **Memory Addressing Modes**: The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations